Università degli Studi di Bologna
Facoltà di Ingegneria

Progettazione di Applicazioni Web T

Esercitazione 3
JDBC e SQL injection
Progettazione persistenza “forza bruta”

Agenda

N.B. Prima di partire, **verificare di aver proceduto con la creazione del proprio SCHEMA sul DB TW_STUD** seguendo le istruzioni presenti nelle note sull'utilizzo di DB2 in LAB4 (rif. pagina 5), sezione Laboratorio sito Web del corso!!

- JDBC e test SQL injection
- Esemplificazione uso API JDBC per progettare la persistenza secondo la metodologia «forza bruta» mediante un **esercizio guidato** relativo alla «*gestione dei dati degli studenti universitari*»
- Proposta di **esercizio da svolgere in autonomia**: «*gestione persistenza di un ordine online rappresentato da un carrello*»

JDBC e test SQL injection

Obiettivo: testare la vulnerabilità a livello di sicurezza dello strato DB di una applicazione

- Importare il progetto Eclipse presente nel file **03_PAWeb.zip** come visto nelle precedenti esercitazioni, senza esploderne l'archivio su file system (lo farà Eclipse)

  File → Import → General → Existing Projects into Workspace → Next → Select archive file

- Riferimento applicazione *test/TestSQLInjection.java*

Operazioni necessarie:

- creare nel proprio schema la tabella `UTENTI`
- popolare le tabelle al punto precedente con alcune tuple di esempio
- testare il codice fornito *TestSQLInjection.java* nella sue due versioni:
  - codice vulnerabile vs. codice non vulnerabile

Test SQL injection: creazione tabella UTENTI

- Creare nel proprio schema la tabella **`UTENTI`**
  (per il momento, procediamo con **interfaccia universale DBeaver o CLP**)

```
UTENTI (
ID char(5) NOT NULL PRIMARY KEY,
PASSWORD char(5) )
```

- Popolare la tabella al punto precedente con alcune tuple di esempio

```
INSERT INTO UTENTI VALUES (… )
```

- Testare a questo punto il codice *TestSQLInjection.java* ricerca nella sua versione vulnerabile e non

- Cosa osserviamo rispetto a quanto detto a lezione? Vulnerabilità di tipo «tautologia»? E rispetto alla vulnerabilità di tipo «istruzione multipla»?

Persistenza con metodologia «forza bruta»

- Ci ricordiamo i passi implementativi della metodologia persistenza JDBC «forza bruta» vero? ☺

- Per ogni ***classe*** **`MyC`** che rappresenta una entità del dominio, si definiscono:
  - un metodo **`doRetrieveByKey(X key)`** che
    - *restituisce un oggetto istanza di* **`MyC`** *i cui dati sono letti dal database* (*tipicamente da una tabella che è stata derivata dalla stessa classe del modello di dominio che ha dato origine a* **`MyC`**)
    - *recupera i dati per chiave*
  - un metodo **`saveOrUpdate(…)`** che *salva i dati dell'oggetto corrente nel database*
    - il metodo *esegue una istruzione SQL update o insert a seconda che l'oggetto corrente esista già o meno nel database*
  - uno o più metodi **`doRetrieveByCond(…)`** che *restituiscono una collezione di oggetti istanza della classe* ***`MyC`*** *che soddisfano una qualche condizione* (basata sui *parametri* del metodo)
  - un metodo **`doDelete(…)`** che *cancella dal database i dati dell'oggetto corrente*

Esercizio guidato

Obiettivo: esemplificare la realizzazione della persistenza secondo la metodologia JDBC «forza bruta» mediante un **esercizio guidato** relativo alla «***gestione dei dati degli studenti universitari***» rappresentati dalla classe del dominio «**Student**»

- scriviamo codice che ha unicamente lo scopo di evidenziare l'uso della API JDBC
- non ci preoccupiamo per il momento della qualità del codice Java

Operazioni

- inserimento di una tupla nel DB
- cancellazione di una tupla
- ricerca di un tupla per chiave primaria
- ricerca di un insieme di tuple (per qualche proprietà)

Lato Java

Consideriamo la classe **`Student`**:

```
package model;
import java.util.Date;

public class Student {
  private int code;
  private String firstName;
  private String lastName;
  private Date birthDate;

   public Student(){}
   public String getFirstName() {
         return firstName;
   }
   public void setFirstName(String firstname) {
         this.firstName = firstName;
   }
   // seguono tutti gli altri metodi getter e setter
}
```

Lato database

…e il DB *tw_stud*:

```
CREATE TABLE students
(
  code INT NOT NULL PRIMARY KEY,
  firstname CHAR(40),
  lastname CHAR (40),
  birthdate DATE
)
```

Ambiente

DBMS

- a scelta tra DB2 (consigliato), HSQLDB e MySQL

Driver JDBC per il DBMS scelto

- per **DB2**: driver **`com.ibm.db2.jcc.DB2Driver`**, contenuto nel file **`db2jcc4.jar`** (scaricabile da https://www.ibm.com/ o dal sito del corso)
- per **HSQLDB**: driver **`org.hsqldb.jdbcDriver`**, contenuto nel file **`hsqldb.jar`** (scaricabile da http://hsqldb.org/ o dal sito del corso)
- per **MySQL**: driver **`com.mysql.jdbc.Driver`**, contenuto nel file **`mysql-connector-java-5.1.x-bin.jar`** (scaricabile da http://dev.mysql.com/downloads/ o dal sito del corso)

Ambiente Java standard: **`.jar/.zip`** del driver deve essere nel **`CLASSPATH`**, ad esempio:

Eclipse Project → Properties → Java Build Path → Libraries → Add JARs

HSQLDB e MySQL

L'ambiente DB2 è già configurato in LAB4; per i DBMS aggiuntivi, occorre operare un breve setup

HSQLDB

- il file *hsqldb.jar* presente nella directory *lib* del progetto contiene già tutto il necessario
- per avviare il server, si consiglia di utilizzare l'apposito target ANT *97.database.start*, oppure *java -cp ../lib/hsqldb.jar org.hsqldb.Server -database.0 file:tw_stud.txt -dbname.0 tw_stud*
- per arrestare il server, avviare il target ANT *98.database.frontend*, inserire in *URL* la stringa *jdbc:hsqldb:hsql://localhost/tw_stud*, eseguire *SHUTDOWN*
- se si vuole si possono scaricare server e connector tramite *http://hsqldb.org → download → hsqldb → hsqldb_2_2 → hsqldb-2.2.0.zip* (il connector si trova nella directory *lib*)

MySQL

- in LAB4 copiare la directory di MySQL disponibile sulla intranet, vedi sezione Laboratorio del sito Web
- il connector si trova nella directory lib del progetto Eclipse
- per avviare il server *bin/mysqld.exe, root* user di default senza password
- **per creare il database**, *bin/mysql.exe -u root → CREATE DATABASE tw_stud;*
- per arrestare il server *bin/mysqladmin.exe -u root shutdown*
- visualizzare database esistenti, *bin/mysql.exe -u root → SHOW DATABASES;*
- da casa scaricare il server tramite *www.mysql.com/donwloads → MySQL Community Server → mysql-5.5.x-win32.zip* e il scaricare il connector tramite *www.mysql.com/downloads → Connector/J → ZIP Archive* e all'interno del file zip si trova il file jar *mysql-connector-java-5.1.x-bin.jar*

Le classi fondamentali di JDBC

Package **`java.sql`** (va importato)

Classe **`DriverManager`**

Interfaccia **`Driver`**

Interfaccia **`Connection`**

Interfaccia **`PreparedStatement`**

Interfaccia **`ResultSet`**

Eccezione **`SQLException`**

Primo passo

- Importare il progetto Eclipse presente nel file **03_PAWeb.zip** come visto nelle precedenti esercitazioni, senza esploderne l'archivio su file system (lo farà Eclipse)

File → Import → General → Existing Projects into Workspace → Next → Select archive file

- Confiniamo nella classe **`DataSource`** le operazioni necessarie per ottenere la connessione
  - il suo compito è fornire connessioni alle altre classi che ne hanno bisogno
  - metodo **`Connection getConnection()`** che restituisce una nuova connessione ad ogni richiesta

È una soluzione artigianale usata solo a fini didattici; verrà raffinata successivamente! ☺

La classe DataSource

```
import java.sql.*;
public class DataSource {
   private String dbURI = "jdbc:db2://diva.disi.unibo.it:50000/tw_stud";
   private String userName = "*******";
   private String password = "*******";

   public Connection getConnection() throws PersistenceException {
    Connection connection;
    try {
        Class.forName("com.ibm.db2.jcc.DB2Driver");
        connection = DriverManager.getConnection(dbURI,userName,
                                                 password);
    } catch (ClassNotFoundException e) {
        throw new PersistenceException(e.getMessage());
    } catch(SQLException e) {
        throw new PersistenceException(e.getMessage());
    }
    return connection;
   }
}
```

! **Nel progetto Eclipse classe *DataSource* gestisce DB2, HSQLDB e MySQL**

Definiamo istruzioni SQL

Vediamo ora il codice JDBC che esegue istruzioni SQL per:

- salvare (rendere persistenti) oggetti nel DB
- cancellare oggetti dal DB
- trovare oggetti dal DB

Si faccia riferimento alla classe **`StudentRepository`**
(concentriamoci in particolare sul codice dei singoli metodi, piuttosto che del progetto di tale classe)

Istruzione SQL: uso di PreparedStatement

Per eseguire una istruzione SQL è necessario creare un oggetto della classe che implementa `PreparedStatement` creato dall'oggetto `Connection` invocando il metodo:

```
PreparedStatement prepareStatement(String s);
```

la stringa `s` è una istruzione SQL parametrica: i parametri sono indicati con il simbolo `?`

Esempio 1

```
String insert = "insert into students (code,
firstname, lastname, birthdate) values (?,?,?,?)";
statement = connection.prepareStatement(insert);
```

Esempio 2

```
String delete = "delete from students where code=?";
statement = connection.prepareStatement(delete);
```

Istruzione SQL: uso di PreparedStatement

I parametri sono assegnati mediante opportuni metodi della classe che implementa **`PreparedStatement`**

- metodi **`setXXX (<numPar>, <valore>)`**

  un metodo per ogni tipo, il primo argomento corrisponde all'indice del parametro nella query, il secondo al valore da assegnare al parametro

Esempio 1 (cont.)

```
PreparedStatement statement;
String insert = "insert into students (code, firstname,
lastname, birthdate) values (?,?,?,?)";
statement = connection.prepareStatement(insert);
statement.setInt(1, student.getCode());
statement.setString(2, student.getFirstName());
statement.setString(3, student.getLastName());

long secs = student.getBirthDate().getTime());
statement.setDate(4, new java.sql.Date(secs));
```

Osservazioni

JDBC usa **`java.sql.Date`**, mentre la classe Student usa **`java.util.Date`**

Le istruzioni

```
long secs = student.getBirthDate().getTime());
statement.setDate(4, new java.sql.Date(secs));
```

servono a "convertire" una data da una rappresentazione all'altra (N.B. per tutti i dettagli si consulti la documentazione)

Esecuzione istruzioni SQL

Una volta assegnati i valori ai parametri, l'istruzione può essere eseguita

Distinguiamo due tipi di operazioni:

- **aggiornamenti** (insert, update, delete): modificano lo stato del database
  - vengono eseguiti invocando il metodo **`executeUpdate()`** sull'oggetto **`PrepareStatement`**
- **interrogazioni** (select): non modificano lo stato del database e restituiscono una sequenza di tuple
  - vengono eseguite invocando il metodo **`executeQuery()`** che restituisce il risultato in un oggetto **`ResultSet`**

Aggiornamenti (insert, delete, update)

Esempio 1 (cont.)

```
PreparedStatement statement;
String insert = "insert into students(code,
firstname, lastname, birthdate) values (?,?,?,?)";
statement = connection.prepareStatement(insert);

statement.setString(1, student.getCode());
statement.setString(2, student.getFirstName());
statement.setString(3, student.getLastName());
long secs = student.getBirthDate().getTime());
statement.setDate(4, new java.sql.Date(secs));

statement.executeUpdate();
```

Interrogazioni (select)

Esempio 2 (cont.)

```
PreparedStatement statement;
String query = "select * from students where code=?";
statement = connection.prepareStatement(query);

statement.setInt(1,code);
ResultSet result = statement.executeQuery();
```

Gestire il risultato di una query

Esempio 2 (cont.)

```
String retrieve = "select * from students where code=?";
statement = connection.prepareStatement(retrieve);
statement.setInt(1, code);
ResultSet result = statement.executeQuery();
if (result.next()) {
   student = new Student();
   student.setCode(result.getInt("code"));
   student.setFirtsName(result.getString("firstname"));
   student.setLastName(result.getString("lastname"));
   long secs = result.getDate("birthdate").getTime();
   birthDate = new java.util.Date(secs);
   student.setBirthDate(birthDate);
}
```

Gestire il risultato di una query

Un altro esempio

```
List<Student> students = null;
Student student = null;
Connection connection = this.dataSource.getConnection();
PreparedStatement statement;
String query = "select * from students";
statement = connection.prepareStatement(query);
ResultSet result = statement.executeQuery();
if(result.next()) {
   students = new LinkedList<Student>();
   student = new Student();
   student.setCode(result.getInt("code"));
   student.setFirstName(result.getString("firstname"));
   …
```

Gestire il risultato di una query

```
    ...
    student.setLastName(result.getString("lastname"));
    student.setBirthDate(
      new java.util.Date(result.getDate("birthdate")
                                          .getTime()));
    students.add(student);
}
while(result.next()) {
    student = new Student();
    student.setCode(result.getInt("code"));
    student.setFirstName(result.getString("firstname"));
    student.setLastName(result.getString("lastname"));
    student.setBirthDate(
      new java.util.Date(result.getDate("birthdate")
                                          .getTime()));
    students.add(student);
}
```

Gestire le eccezioni

- Tutti i metodi delle classi dell'API JDBC visti fino ad ora "lanciano" una eccezione **`SQLException`**
- *Connection*, *statement*, *ResultSet* devono essere **sempre "chiusi"** dopo essere stati usati
  - l'operazione di chiusura corrisponde al rilascio di risorse
- Effettuiamo queste operazioni nella clausola **`finally`**
  - abbiamo così la garanzia che vengano comunque effettuate
- Se vengono sollevate eccezioni, le rilanciamo con una eccezione di livello logico opportuno
- Il codice che segue è un po' verboso, ma didatticamente ci aiuta a ragionare ancora sulla gestione delle eccezioni

Clausola `finally`

```
finally {
   try {
       if (statement != null)
          statement.close();
       if (connection!= null)
          connection.close();
   }
   catch (SQLException e) {
       throw new PersistenceException(e.getMessage());
   }
}
```

Ora a voi (1)

Prendendo spunto **dall'esercizio guidato** appena mostrato, e utilizzando il DBMS DB2, si realizzi l'applicazione java «**Negozio online**» in grado di **gestire la persistenza di un ordine usando la metodologia Forza Bruta**

Nel dettaglio, l'applicazione si basa sui seguenti Java Bean:

- **Cart**, implementato con il codice cart (**int CodCart**) e un oggetto di tipo **Map<Item, Integer>**, che associa ad un **Item** la quantità ordinata di tale Item e inserita dall'utente nel suo carrello. Inoltre, ogni carrello è caratterizzato **dall'email dell'acquirente** (**String email**)
- **Item**, mantiene il codice item (**int CodItem**) la descrizione dell'item (**String description**), il suo prezzo (**double price**) e la sua disponibilità in catalogo (**int quantity**)
- **Catalogue**, è unico e realizzato come **List<Item>**

Ora a voi (2)

Dopo aver **creato da applicazione Java lo schema delle tabelle** necessarie a mantenere i dati dei Java Bean, **implementato i Java Bean** e **metodi necessari per la realizzazione delle operazioni CRUD**, si crei una classe «**ShopRepository**» in grado di

a) rendere persistente un ordine

b) offrire la possibilità di ottenere gli ordini relativi ad una email

c) offrire la possibilità di recuperare tutti gli ordini suddivisi per email

N.B.1 Focalizzarsi unicamente sulla **gestione dell'ordine** di un utente (e non sulla gestione del catalogo, e dunque del decremento del numero di oggetti presenti nel catalogo)

N.B.2 Non c'è un mapping 1-1 tra Java Bean e tabelle! In questo caso, ad esempio, c'è un Java Bean che usa una tabella di un altro Java Bean... Quale?